A Formal Definition of SOL

D. E. KNUTH AND J. L. McNELEY

Summary—This paper gives a formal definition of SOL, a general-purpose algorithmic language useful for describing and simulating complex systems. SOL is described using meta-linguistic formulas as used in the definition of ALGOL 60. The principal differences between SOL and problem-oriented languages such as ALGOL or FORTRAN is that SOL includes capabilities for expressing parallel computation, convenient notations for embedding random quantities within arithmetic expressions and automatic means for gathering statistics about the elements involved. SOL differs from other simulation languages such as SIMSCRIPT primarily in simplicity of use and in readability since it is capable of describing models without including computer-oriented characteristics.

I. General Description

SOL IS an algorithmic language used to construct models of general systems for simulation in a readable form. The model builder describes his model in terms of processes whose number and detail are completely arbitrary and definable within the constraints of the language elements. A SOL model consists of a number of statements and declarations which have a character similar to that found in programming languages such as ALGOL.

The model is not built to be executed in a sequential fashion as ordinary programming languages require. Rather, the processes are written and executed as if all were running in parallel. Control between processes is maintained by the interaction of *global* entities and by control and communication instructions within the different processes. At the initiation of the simulation all processes are begun simultaneously.

Variables declared within a process are called *local* variables. Within a given process it is possible to have several actions going on at once; therefore, we may think of several objects on which the action takes place each in its own place in the process at any given time. These objects will be referred to as *transactions*. A set of local variables corresponding in number to those declared in the process is "carried with" each transaction of that process. Transactions situated within one process may not refer to the local variables of another process nor to the local variables of another transaction in the same process.

Global quantities are of three major types: global variables, facilities and stores. *Global variables* can be referenced or changed by any transaction from any process in the system, and the variable possesses only one value at any given time.

Manuscript received January 3, 1964.

D. E. Knuth is with the California Institute of Technology, Pasadena, Calif.

J. L. McNeley is with the Burroughs Corporation, Pasadena, Calif.

A *facility* is a global element which can be controlled by only one transaction at a time. Associated with each request for the facility is a "control strength," and if a requesting transaction has a higher strength than the transaction controlling the facility, an interrupt will occur. Interrupts may be nested to any depth. If the requesting transaction is not of greater strength than the controlling transaction, then the requesting transaction stops and waits for the facility until the controlling transaction releases its control. When a transaction is interrupted, it cannot advance to any other position in its program until it regains control of the facility.

Stores are space-shared rather than time-shared global elements, and they are assigned a specific storage capacity. As long as there is sufficient storage to accommodate the requesting transaction the request for space is satisfied; otherwise, the transaction waits until the space it is requesting becomes available. In this sense, a facility may be regarded as a store which has a capacity of one unit only, except for the fact that no interrupt capability is provided for stores.

Simulated time passes in discrete units indicated in "wait statments." The model builder requires the transactions to wait a proper number of time units at the appropriate places in the processes, and this specifies the time element. The interpretation of the physical significance of a unit of time is immaterial in the SOL language; if all time interval specifications are multiplied by a factor of ten it will not decrease the speed by which the model is simulated.

Control within or between processes is also introduced into the simulation by allowing a transaction to wait until a global variable or expression obtains a certain value. A transaction may also be forced to wait until a space- or time-shared element attains a certain status.

Output statements which display the progress of the simulation may be inserted at will in the model. Special types of statistics are automatically available, such as the per cent of utilization of a facility, the average and maximum number of elements in a store at a given moment, etc. Another type of global quantity, called a *table*, is introduced to record statistical information about desired data. The mean, the standard deviation and a histogram are provided for all data recorded in a table.

Processes initiate parallelism within themselves by using a duplication operation. The transaction makes an exact copy of itself and sends the copy to a specified location in the process while the original continues in sequence. A transaction is taken out of the system when it executes a "cancel" statement.

Other operations available in SOL are similar to those of existing algorithmic languages, but these portions of the language are at the present time less powerful than the features available in a large scale programming language.

A detailed example of a complete SOL model appears in a companion paper in this issue [2].

II. Syntax and Semantics of SOL

We will define the syntax of SOL using meta-linguistic formulas as given in the definition of ALGOL 60 [1]. Certain things which have been carefully defined in ALGOL 60 will not be redefined here but will merely be stated to have the same interpretation as given by ALGOL. We will use the abbreviation *⟨A⟩* to mean "a list of ⟨A⟩," *i.e.*,

$$*\langle A\rangle*::=\langle A\rangle \mid *\langle A\rangle*, \langle A\rangle$$

Comments may be written in the form "**comment** ⟨string without semicolons⟩;" as in ALGOL 60.

A. Identifiers and Constants

⟨letter⟩::=A|B|C|D| · · · |Z
⟨digit⟩::=0|1|2|3| · · · |9
⟨number⟩::=⟨constant⟩|⟨decimal constant⟩
⟨constant⟩::=*⟨digit⟩*
⟨decimal constant⟩::=⟨constant⟩.⟨constant⟩
⟨identifier⟩::=⟨letter⟩|⟨identifier⟩⟨letter⟩|
⟨identifier⟩⟨digit⟩

Identifiers are used as the names of variables, statistical tables, stores, facilities, processes, procedures and statements. The same identifier can be used for only *one* purpose in a program. Constants are used to represent integer numbers. Decimal constants represent real numbers. Identifiers must be declared before they are used elsewhere.

B. Declarations

⟨declared item⟩::=⟨identifier⟩|⟨identifier⟩[⟨constant⟩]
⟨variable declaration⟩::=**integer** *⟨declared item⟩*|
real* ⟨declared item⟩*
⟨facility declaration⟩::=**facility** *⟨declared item⟩*
⟨store declaration⟩::=**store** *⟨constant⟩⟨declared item⟩*
⟨table declaration⟩::=**table** *(⟨number⟩**step**⟨number⟩
until ⟨number⟩)⟨declared item⟩*
⟨monitor declaration⟩::=**monitor** *⟨identifier⟩*

If the declared item is simply an identifier, it means that a single item of that name is being declared. The other form, *e.g.*, A[10], means 10 similar items called A[1], A[2], · · · , A[10] are being declared.

The variable declaration is used to specify variables (either local or global, depending on where the declaration appears). All variables are initially set to zero when declared. "Integer" variables differ from "real" variables in that when a value is assigned to them it is rounded to the nearest integer.

When a facility is declared, it is initially "not busy"; at the end of the simulation run, statistics are reported giving the per cent of time each facility was in use.

A store declaration gives the capacity of each store (the number preceding the identifier). At the end of the simulation run statistics are given on the average and the maximum number of items occupying the store (as a function of time). Stores are empty when first declared.

A "table" is used to gather detailed statistical information of any desired type; readings are tabulated and afterwards the mean, the standard deviation, histogram distribution, etc., are output. The constants preceding the table name give the starting point for histogram intervals, the increment between intervals and the highest value.

A monitor declaration names items which already have been declared, with the understanding that these identifiers are to be "monitored." This means that whenever a change in the state of the corresponding quantity is detected, a line will be printed giving the details. This capability is especially useful when checking out a model, and it can also be used to advantage for output during a regular simulation run.

C. Expressions and Relations

⟨name⟩::=⟨identifier⟩|⟨identifier⟩[⟨expression⟩]

By ⟨variable name⟩, ⟨facility name⟩, etc., we will mean that the identifier in the name has appeared in a ⟨variable declaration⟩, ⟨facility declaration⟩, etc., respectively.

⟨primary⟩::=⟨variable name⟩|⟨store name⟩|
⟨constant⟩|⟨decimal constant⟩|**time**|
(*⟨expression⟩*)|abs(⟨expression⟩)|
max(*⟨expression⟩*)|min(*⟨expression⟩*)|
normal(⟨expression⟩, ⟨expression⟩)|
exponential(⟨expression⟩)|poisson(⟨expression⟩)|
geometric(⟨expression⟩)| **random**
⟨term⟩::=⟨primary⟩|⟨term⟩×⟨primary⟩|
⟨term⟩÷⟨primary⟩|⟨term⟩/⟨primary⟩|
⟨term⟩**mod**⟨primary⟩
⟨sum⟩::=⟨term⟩|+⟨term⟩|−⟨term⟩|⟨sum⟩+⟨term⟩|
⟨sum⟩−⟨term⟩
⟨unconditional expression⟩::=⟨sum⟩|⟨sum⟩:⟨sum⟩
⟨expression⟩::=⟨unconditional expression⟩|
if ⟨relation⟩ **then** ⟨expression⟩ **else** ⟨expression⟩

The meaning of the arithmetical operations inside expressions is identical to the meaning in ALGOL 60.

The new elements here are "*a* **mod** *b*," the positive remainder obtained upon dividing *a* by *b*; "max(e_1, · · · , e_n)" and "min(e_1, · · · , e_n)," which denote the maximum and minimum values, respectively, of the *n* expressions; and there are also notations for expressing random values. The expression "(e_1, · · · , e_n)" indicates that a random selection is made from among the *n* expressions with equal probability of choosing any

expression. The expressions normal(M, S), poisson(M), geometric(M) and exponential(M) indicate random values with special distributions which occur frequently in applications. A random number drawn from the normal distribution with mean M and standard deviation S is denoted by normal(M, S) and is a real (not necessarily integer) value. A number drawn from the exponential distribution with mean M is denoted by exponential(M) and is also of type real. The poisson distribution signified by poisson(M), on the other hand, yields only integer values; the probability that poisson(M) = n is $(e^{-M}M^n/n!)$. The geometric distribution with mean M, denoted by geometric(M), also yields integer values, where the probability that geometric(M) = n is $1/M(1-1/M)^{n-1}$. The symbol **random** denotes a random real number between 0 and 1 having uniform distribution. Finally, we have the notation $e_1{:}e_2$, which denotes a random integer between the limits e_1 and e_2; more formally

$$e_1{:}e_2 = \begin{cases} 0, & e_1 > e_2 \\ (e_1, e_1+1, \cdots, e_2) & e_1 \leqq e_2. \end{cases}$$

The normal, exponential, poisson and geometric distributions are mathematically expressible in terms of **random** as follows:

$$\text{normal(M,S)} = S \times \sqrt{-2 \ln (\textbf{random})} \times \sin (2\pi\, \textbf{random}) + M$$

$$\text{exponential(M)} = -M \ln (\textbf{random})$$

$$\text{poisson(M)} = n \text{ if } e^{-M}\left(1 + M + \frac{M^2}{2!} + \cdots + \frac{M^{n-1}}{(n-1)!}\right) \leqq \textbf{random} < e^{-M}\left(1 + M + \cdots + \frac{M^n}{n!}\right)$$

$$\text{geometric(M)} = \left[1 + \ln (\textbf{random})/\ln\left(1 - \frac{1}{M}\right)\right].$$

(The poisson distribution should not be used for values of M greater than 10.) As examples of the use of these distributions, consider a population of customers coming to a market with an average of one customer every M minutes. The distribution of waiting time between successive arrivals is exponential(M). On the other hand, if an average of M customers come in per hour, the distribution of the actual number of customers arriving in a given hour is poisson(M). If an individual performs an experiment repeatedly with a chance of success, 1/M on each independent trial, the number of trials needed until he first succeeds is geometric(M).

The special symbol "**time**" indicates the current time; intially, **time** is zero. The value of a store name is the current number of occupants of the store.

⟨relational operator⟩::= = | ≠ | < | ≦ | > | ≧
⟨relation primary⟩::= ⟨unconditional expression⟩
⟨relational operator⟩⟨unconditional expression⟩ |
⟨facility name⟩ **busy** | ⟨facility name⟩ **not busy** |
⟨store name⟩ **full** | ⟨store name⟩ **not full** |
⟨store name⟩ **empty** | ⟨store name⟩ **not empty** |
pr(⟨expression⟩) | (⟨relation⟩)
⟨relation⟩::= ⟨relation primary⟩ |
⟨relation primary⟩∨⟨relation primary⟩ |
⟨relation primary⟩∧⟨relation primary⟩ |
¬⟨relation primary⟩

These relations have obvious meanings except for the construction "pr(e)" which stands for a random condition which is true with probability e. (Here e must be less than or equal to 1.) Thus we might say

if pr(0.12) **then** (12 per cent of the time)
else (88 per cent of the time).

III. Statements

A. *Processes*

As this simulator operates, any number of processes written in the language may be in use at once. We may think of several objects, each in its own place in the process at any given time. These objects are referred to as *transactions*. In this section, we describe the various manipulations that transactions can perform in the language.

⟨process description⟩::= **process** ⟨identifier⟩;
⟨statement⟩ |
process ⟨identifier⟩; **begin**
⟨process declaration list⟩; ⟨statement list⟩ **end**
⟨process declaration⟩::= ⟨variable declaration⟩ |
⟨procedure declaration⟩ | ⟨monitor declaration⟩
⟨process declaration list⟩::= ⟨process declaration⟩ |
⟨process declaration list⟩; ⟨process declaration⟩

There are two kinds of variables, *global* variables (not declared in a process) and *local* variables (those which are declared in a process). All transactions can refer to the global variables, and a global variable has only one value at any given time. But a local variable is "carried with" each transaction within a given process, and there is in general, a different value for a local variable depending on which transaction is using it. Transactions situated within one process may not refer to the local variables of another process, nor can the local variables of one transaction within a process be reached directly by other transactions in that same process. Communication between processes is accomplished solely with the help of global quantities.

B. *Labels*

A statement may be named by any identifier as follows:

⟨statement⟩::= ⟨unlabeled statement⟩ |
⟨identifier⟩:⟨statement⟩

By the designation ⟨label⟩ we will mean the name of a statement.

C. Creation of Transactions

At the beginning of simulation, there is one transaction present for each process described. Each of these initial transactions starts at time zero and is positioned at the beginning of the process. More transactions may be created by using "start statements."

⟨start statement⟩::=**new transaction to** ⟨label⟩

This statement, when executed, creates a new transaction (whose local variables are the same in number and value as those of the transaction which created it). The new transaction begins executing the program at ⟨label⟩ while the original transaction continues in sequence. New transactions are also created by input statements (Section III-T).

D. Disappearance of Transactions

Transactions "die" when they execute a cancel statement.

⟨cancel statement⟩::=**cancel**

An implied cancel statement is at the end of every process, so cancel statements need not always be explicitly written.

E. Replacement Statements

⟨replacement statement⟩::=⟨variable name⟩ ←⟨expression⟩

This replaces the value of the variable by the value of the expression. The variable may be global or local, but not the name of a store. If the variable is an integer variable, the expression is rounded.

F. Priority

Time is measured in discrete units, so it may happen that by coincidence two transactions want to do something at precisely the same time. They may be in conflict, *e.g.*, they may both want to seize a facility, or to change the value of the same global variable or one may want to change it while the other is using its value. Actually, in such cases of conflict, the simulator does choose a specific order for execution; no two things actually happen at the same instant, as we deal more properly with *infinitesimal* units of time between the discrete units. The choice of order is fairly arbitrary except when a difference of priority is specified; in that case, the transaction with *higher* priority will be acted on first. Each transaction has a priority, which is initially zero; priority is changed by the statement

PRIORITY←⟨expression⟩.

The declaration "**integer** PRIORITY" is implied at the beginning of each process, *i.e.*, PRIORITY is treated as a local variable. In the present implementation of SOL, the priority must be between 0 and 63. The effect of priority is spelled out further in Section IV.

G. Wait Statements

⟨wait statement⟩::=**wait** ⟨expression⟩

The expression is rounded to the nearest integer, and then this statement advances "**time**" by max(0, ⟨expression⟩), as far as this transaction is concerned. All time delays in a simulated process are, in the last analysis, specified by using wait statements.

H. Wait-Until Statements

⟨wait-until statement⟩::=**wait until** ⟨relation⟩

This causes the transaction to freeze at this point until the relation becomes true (because of action by other transactions). The relation must not involve expressions which have a random value; *e.g.*, it is not legal to write "**wait until** pr(10)" or "**wait until** A[1:4]=0," etc.

I. Enter Statements

⟨enter statement⟩::=**enter** ⟨store name⟩|
enter ⟨store name⟩, ⟨expression⟩

The first form is an abbreviation for "**enter** ⟨store name⟩, 1." The value of the expression, rounded to the nearest integer, gives the number of units requested of the store. The transaction will remain at this statement until that number of units is available and until all other transactions of greater or equal priority which have been waiting for storage space have been serviced.

J. Leave Statement

⟨leave statement⟩::=**leave** ⟨store name⟩|
leave ⟨store name⟩, ⟨expression⟩

The first form is an abbreviation for "**leave** ⟨store name⟩, 1." This statement returns the number of units equivalent to the value of the (rounded) expression.

K. Seize Statements

⟨seize statement⟩::=**seize** ⟨facility name⟩|
seize ⟨facility name⟩, ⟨expression⟩

The first form is equivalent to "**seize** ⟨facility name⟩, 0." This statement is usually rather simple, but there are situations when complications arise. If the facility is not busy when this statement occurs, then it becomes busy at this point and remains busy until later released by this transaction. (Note: If this transaction creates another transaction by means of a start statement, the new transaction does not control the facility.)

The expression appearing above represents the "control strength" which is normally zero. Allowance is made, however, for one transaction to interrupt another. If the facility is busy when the seize statement occurs, let E_1 be the control strength with which the facility was seized and let E_2 be the control strength of this seize statement. If $E_2 \leqq E_1$, the transaction waits until the facility is not busy. If $E_2 > E_1$, however, *interrupt* occurs. The transaction T_1 which had control of

the facility is stopped wherever it was in its program, and the present transaction T_2 seizes the facility. When T_2 releases the facility, the following occurs:

1) If T_1 was executing a wait statement when interrupted, the time of wait is increased by the time which passed during the interrupt.
2) There may be several transactions not waiting to seize this facility. If any of these has a higher control strength than E_1, then T_1 is interrupted again. The transaction which interrupts is chosen by the normal rules for deciding who obtains control of a facility upon release, as described in the next section.

The control strength in the present implementation of SOL must be an integer between 0 and 4095. This allows interrupts to be nested up to 4095 deep.

L. Release Statements

⟨release statement⟩::=**release** ⟨facility name⟩

This statement is permitted only when the transaction is actually controlling the facility because of a previous seizure. When the facility is released, there may be several other transactions waiting because of seize statements. In this case, the one which gets control of the facility next is chosen by a consideration of the following three quantities in order:

1) highest control strength,
2) highest PRIORITY,
3) first to request the facility.

M. Go To Statements

⟨go to statement⟩::=**go to** ⟨label⟩|
go to (*⟨label⟩*), ⟨expression⟩

This statement is used to transfer to another point in the program; statements are usually executed sequentially. In the second form, the expression is used to select which statement to transfer to; if there are n labels, the expression, when rounded to the nearest integer, must have a value between 0 and n. Zero means continue in sequence, 1 means go to the first statement mentioned, and so on.

N. Compound Statements

Several statements may be combined into one, as follows:

⟨statement list⟩::=⟨statement⟩|⟨statement list⟩;
⟨statement⟩
⟨compound statement⟩::=**begin** ⟨statement list⟩ **end**|
(⟨statement list⟩)

O. Conditional Statements

⟨conditional⟩::=**if** ⟨relation⟩ **then** ⟨unconditional statement⟩|
if ⟨relation⟩ **then** ⟨unconditional statement⟩ **else** ⟨statement⟩

The meaning is the same as in ALGOL; testing of the relation requires no simulated time.

P. Tabulate Statements

⟨tabulate statement⟩::=**tabulate** ⟨expression⟩ **in** ⟨table name⟩

The value of the expression is recorded as a statistical observation in the table specified.

Q. Output Statements

⟨carriage control⟩::=⟨empty⟩|**page**|**line**|**double**
⟨string⟩::=⟨any sequence of characters excluding "#"⟩
⟨output list item⟩::=#⟨string⟩#|⟨expression⟩|
⟨store name⟩|⟨table name⟩|⟨facility name⟩
⟨output statement⟩::=**output** *⟨carriage control⟩
⟨output list item⟩*

Output occurs for all items listed, in turn, after doing the appropriate carriage control positioning. The output for a string is the string itself. An output for an expression is the value. For a store, table or facility, the appropriate statistical information is output. At the conclusion of an output statement, the final line is printed out.

R. Stop Statements

⟨stop statement⟩::=**stop**

A stop statement causes simulation to terminate immediately, and all transactions cease. The statistics for all stores, tables and facilities are output as in the output statement, as well as the final time, the number of times each labeled statement was referenced and the number of transactions which appeared in each process.

S. Procedures

⟨procedure declaration⟩::=**procedure** ⟨identifier⟩;
⟨statement⟩
⟨procedure statement⟩::=⟨procedure name⟩

A procedure is simply a subroutine used to save coding. Parameters are not allowed, but their effect can be achieved by setting local variables in the transactions before calling the procedure. There are local procedures and global procedures (the latter are declared outside of a process). Global procedures cannot refer to local variables. A go to statement may not lead out of a procedure body. Procedures may be used recursively.

T. Transaction Input-Output

⟨transaction read statement⟩::=**read** ⟨constant⟩ **to** ⟨label⟩
⟨transaction write statement⟩::=**write** ⟨constant⟩

The read statement inputs a set of values of local variables for a transaction of the same type as the one executing the read statement; this set of values is used in the creation of a new transaction which begins exe-

cuting the program at the statement mentioned. The write statement writes the current values of the local variables of the transaction onto the unit specified and does not cancel the present transaction. The constant in each refers to a tape or card unit number. The same tape should not be used for both input and output in the same simulation run.

U. Summary of Statements

⟨unlabeled statement⟩::=⟨unconditional statement⟩|
⟨conditional⟩
⟨unconditional statement⟩::=⟨start statement⟩|
⟨cancel statement⟩|
⟨replacement statement⟩|⟨wait statement⟩|
⟨wait-until statement⟩|⟨enter statement⟩|
⟨leave statement⟩|⟨seize statement⟩|
⟨release statement⟩|⟨go to statement⟩|
⟨compound statement⟩|⟨output statement⟩|
⟨tabulate statement⟩|⟨stop statement⟩|
⟨transaction read statement⟩|⟨procedure statement⟩|
⟨transaction write statement⟩|⟨empty⟩

IV. The Model as a Whole

⟨model⟩::=**begin** ⟨global declaration list⟩; ⟨process list⟩
end.
⟨declaration⟩::=⟨variable declaration⟩|
⟨facility declaration⟩|
⟨store declaration⟩|⟨table declaration⟩|
⟨monitor declaration⟩|⟨procedure declaration⟩
⟨global declaration list⟩::=⟨declaration⟩|
⟨global declaration list⟩; ⟨declaration⟩
⟨process list⟩::=⟨process sdescription⟩|
⟨process list⟩; ⟨process description⟩

Initially all variables are zero, all facilities are "not busy," all stores are "empty," the time is zero, one transaction appears for each process described and the simulator is in the "choice state."

When the simulator is in "choice state," each transaction is either positioned at a wait statement, a wait-until statement, a seize or enter statement or else it has just been created. (We will dispense with the latter case by assuming a "wait 0" statement has been inserted just before the present position when a new transaction is created.) If there are no transactions which can move at this time, the time is advanced to the earliest completion time for a wait statement. Now, from the set of transactions able to move, that one is selected which has the highest PRIORITY, and in case of ties, which has been waiting the longest. (If there is still a tie, an arbitrary choice is made.) The selected transaction is activated, and it continues to execute its statements until encountering a cancel or stop statement, a priority assignment statement, a wait statement, a wait-until statement with a false relation or a seize or enter statement which cannot take place at that time. We examine all other transactions which are stopped because of a wait-until statement involving global quantities changed by the present transaction. If the corresponding relation is now true, these transactions become free to move at the current time. Then we have once again reached "choice state." Note that all release statements which are passed during the time the selected transaction was moving are processed immediately in such a way that the facility becomes not busy only if no other transaction were interrupted or were waiting to seize it; if other transactions *are* in the latter category, the choice of successor and the transfer of control described in Section III-L takes place immediately as the release statement is executed. Therefore, it is conceivable that the statement "wait until FAC not busy" may never be passed if other transactions are always ready to seize the facility FAC. Similar remarks apply to the leave statements.

Since this paper was written, a few additions have been made to the SOL language, including "synchronous" variables and some additional diagnostic capabilities.

References

[1] "Revised report on the algorithmic language ALGOL 60," *Comm. ACM*, pp. 1–17; January 6, 1963.
[2] D. E. Knuth and J. L. McNeley, "SOL—A symbolic language for general-purpose systems simulation," this issue, page 401.